PATUS for Convenient High-Performance Stencils: Evaluation in Earthquake Simulations

Matthias Christen
Institute of Computational Science,
Faculty of Informatics,
University of Lugano,
Switzerland
matthias.christen@usi.ch

Olaf Schenk
Institute of Computational Science,
Faculty of Informatics,
University of Lugano,
Switzerland
olaf.schenk@usi.ch

Yifeng Cui
San Diego Supercomputer Center,
USA
yfcui@sdsc.edu

***Abstract*—PATUS is a code generation and auto-tuning framework for stencil computations targeting modern multi and many-core processors. The goals of the framework are productivity and portability for achieving high performance on the target platform. Its stencil specification language allows the programmer to express the computation in a concise way independently of hardware architecture-specific details. Thus, it increases the programmer productivity by removing the need for manual low-level tuning. We illustrate the impact of the stencil code generation in seismic applications, for which both weak and strong scaling are important. We evaluate the performance by focusing on a scalable discretization of the wave equation and testing complex simulation types of the AWP-ODC code to aim at excellent parallel efficiency, preparing for petascale 3-D earthquake calculations.**

I. INTRODUCTION

Stencil computations are constituent computational building blocks in many scientific and engineering codes. These codes typically achieve a low fraction of peak performance. Computational domains that involve stencils include medical and life science applications, petroleum reservoir simulations, weather and climate modeling, and physics simulations such as fluid dynamics or quantum chromodynamics. Stencil codes may perform tens of thousands of iterations over the spatial domain in order to resolve the time-dependent solution accurately; hence, may require significant core hours on supercomputers. Thus, any performance improvement may lead to a significant reduction in time to solution.

Despite the apparent simplicity of stencil computations and the fact that their computation structure maps well to current hardware architectures, meticulous architecture-specific tuning is still required to elicit a platform's full computational power due to the fact that microarchitectures have grown increasingly complex. Manual architecture-specific tuning requires a significant effort: Not only does it require a deeper understanding of the architecture, but it is also both a time consuming and error-prone process. Although the performance gain might justify the effort, the code usually becomes nonportable and hard to maintain. The goal of PATUS is to accept an intuitive stencil specification and turn it into architecture-specific, optimized, high-performance code.

The finite difference method based AWP-ODC simulates dynamic earthquake rupture and wave propagation. For these kinds of applications, the finite difference method is a good trade-off between accuracy, computational efficiency, and ease of implementation.

AWP-ODC has been widely used by researchers at the South California Earthquake Center (SCEC) for community large-scale wave propagation simulations. The SCEC code is also used for full 3-D tomography to refine Earth structures, and reciprocity-based simulations to create a statewide (California) physics-based seismic hazard map—a project requiring hundreds of millions of core-hours calculations in the years to come.

AWP-ODC was an ACM Gordon Bell finalist at SC'10: on the Oak Ridge National Laboratory XT5 "Jaguar," the code achieved "M8," a full dynamic simulation of a magnitude-8 earthquake on the southern San Andreas fault, at a maximum frequency resolution of 2 Hz. This production run, which produced 360 seconds of wave propagation, sustained 220 TFlop/s for 24 hours using 223,074 Jaguar cores [1].

In this work, we contribute to the domain of stencil computations, with respect to the cited previous work, in the following areas:

- We leverage domain specific language (DSL) techniques, domain specific code generation, and the auto-tuning methodology, i.e., executing automated benchmarks guided by a search method to select the code variant with the best performance, to make stencil kernels both code- and performance-portable across hardware platforms. These methodologies and techniques are wrapped into our tool, PATUS, which stands for Parallel Auto-TUned Stencils [2], [3].
- We present a careful performance study of various compilers and analyze the performance of the automatic stencil code generation against them and show scalability on the current microarchitectures of AMD and Intel.
- We apply PATUS to a case study solving a fourth-order space and second-order time finite-difference discretization of the wave equation and show the impact of the code generation on up to 1024 Cray XE6 nodes.
- We use PATUS to generate and tune codes for stencils originating from a scalable real-world earthquake simula-

SC12, November 10–16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

tion application AWP-ODC [1], a highly scalable parallel finite-difference application, that enables petascale 3-D earthquake calculations and aims for excellent parallel efficiency. The code has been developed at the Southern California Earthquake Center and has been used to conduct many of SCEC's major earthquake simulations.

The rest of this paper is organized as follows: In section II, stencil computations and stencil-specific performance challenges are discussed. In section IV the PATUS framework is presented, which is followed by a section on performance and scalability results both for single stencils (on both an AMD and on an Intel platform) and two full applications: a wave equation solver and the AWP-ODC code.

II. STENCIL COMPUTATIONS

A stencil computation is characterized by updating each point in a structured grid by an expression depending on the values on a fixed geometrical structure of neighboring grid points. In our context, we assume that the grid is rectilinear (or has a Cartesian representation), which is the case in most stencil-based codes.

We give a few examples of stencils to highlight some of their variety. We will also present performance benchmarks in section V for these examples.

(1) The discrete 3-D Laplacian:

$$u'_{ijk} = \alpha u_{ijk} + \beta \left(u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1} \right).$$

This is a second-order finite difference discretization of the continuous Laplacian $\Delta := \sum_i \frac{\partial^2}{\partial x_i^2}$. Figure 1(a) shows a visualization of the stencil structure.

(2) A higher-order discretization of the 3-D Laplacian ("Laplacian GSRB"):

$$u'_{\mathbf{i}} = a u_{\mathbf{i}} + b \sum_{|\mathbf{j}|_1=1} u_{\mathbf{i}+\mathbf{j}} + \sum_{r=1}^{2} c_r \sum_{\substack{|\mathbf{j}|_1=2,\\ \mathbf{j}_k\in\{0,\pm r\},\\ k=1,2,3}} u_{\mathbf{i}+\mathbf{j}}, \quad \mathbf{i},\mathbf{j}\in\mathbb{Z}^3,$$

where $\mathbf{j} = (\mathbf{j}_1, \mathbf{j}_2, \mathbf{j}_3) \in \mathbb{Z}^3$ is an offset vector. If the parameters are set to $a = \alpha + 6\beta^2$, $b = \alpha\beta$, $c_1 = 2\beta^2$, $c_2 = \beta^2$, this stencil corresponds to two iterations of a red-black Gauss-Seidel iteration collapsed into one. Such Gauss-Seidel Laplacians are frequently used as smoothers in multigrid methods. The structure of this stencil is shown in Figure 1(b).

(3) A stencil which comes from discretizing the classic wave equation $\frac{\partial^2}{\partial t^2} u - c^2 \Delta u = 0$ (with appropriate initial and boundary conditions):

$$u_{ijk}^{(t+1)} = \left(2 + d_0 \left(\tfrac{\delta t}{\delta x}\right)^2 c^2\right) u_{ijk}^{(t)} - u_{ijk}^{(t-1)} + \left(\tfrac{\delta t}{\delta x}\right)^2 c^2 \sum_{r=1}^{2} d_r \left(u_{i\pm r,j,k}^{(t)} + u_{i,j\pm r,k}^{(t)} + u_{i,j,k\pm r}^{(t)} \right)$$

with $d_0 = \frac{-15}{2}$, $d_1 = \frac{4}{3}$, $d_2 = \frac{-1}{12}$. The stencil comes from a fourth-order-in-space and second-order-in-time discretization of the wave equation. This stencil, in contrast to the previous examples, depends on two previous time steps. Its structure is visualized in Figure 1(c).

(4) The two-dimensional Gaussian blur filter stencil:

$$u'_{i,j} = \sum_{|(\iota,\eta)|_\infty \le 2} G(\iota,\eta) \cdot u_{i+\iota,j+\eta}$$

with $G(\iota,\eta) := \frac{1}{2\pi\sigma^2} \exp\left(-\frac{\iota^2+\eta^2}{2\sigma^2}\right)$. Such a stencil is often used in image processing applications; this one, in particular, to reduce image noise. The parameter σ is the standard deviation of the Gaussian distribution. A visualization of this stencil can be found in Figure 1(d).

(5) Two stencils occurring in the anelastic wave propagation code AWP-ODC. It solves the 3-D velocity-stress wave equations using an explicit method with a staggered-grid finite difference method which is fourth-order accurate in space and second-order accurate in time. The code is based on the finite difference code originally developed by Olsen [4]. The code primarily entails updating two coupled quantities in space and in time: the model's governing equations is the system

$$\frac{\partial \mathbf{v}}{\partial t} = \tfrac{1}{\rho} \nabla \cdot \boldsymbol{\sigma}, \quad \frac{\partial \boldsymbol{\sigma}}{\partial t} = \lambda (\nabla \cdot \mathbf{v}) \mathbf{I} + \mu (\nabla \mathbf{v} + \nabla \mathbf{v}^\intercal)$$

of PDEs. The dependent variables are the velocity vector field $\mathbf{v}$ and the (symmetric) stress tensor $\boldsymbol{\sigma}$. λ and μ are the Lamé coefficients, ρ is the density, and $\mathbf{I}$ is the identity tensor.

The Perfectly Matched Layers approach, damping separately for wavefields propagating parallel and perpendicular to the boundary, is used on the sides and at the bottom of the grid, and a zero-stress free surface boundary condition is used at the top [5].

The first stencil kernel ("UXX") updates a component of the velocity vector from a given stress tensor, and the second stencil ("ALLQ") updates the viscous stress tensor based on the given velocity field.

A. The Challenge of Achieving Good Performance

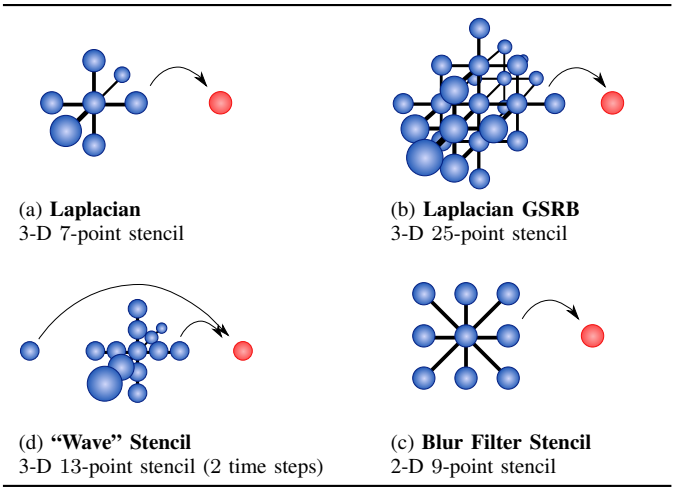
From a hardware architecture point of view, stencil computations on structured grids are attractive because of the regularity of the data access pattern. This allows using the hardware prefetcher in an optimal way and streaming in the data from the main memory to the compute elements.

Yet, a typical stencil computation performs only a very limited number of floating point operations per grid point; the numbers for the example stencils are shown in Table I in the "FLOPs" column. Hence, stencils typically have a low *arithmetic intensity*, i.e., a low FLOP-per-transferred-Byte ratio. Table I shows the number of grids that need to be read and written, which asymptotically is the number of data elements that have to be transferred per stencil computation. The low arithmetic intensities imply that the performance of typical stencil computations is limited by the available bandwidth to the memory subsystem.

In order to achieve good performance, it is critical to minimize noncompulsory data transfers and reuse available data as efficiently as possible. Jacobi iterations have the advantage

TABLE I
CHARACTERISTICS OF THE EXAMPLE STENCILS (SINGLE PRECISION)

	Stencil	FLOPs	Data Streams (Number of in/out grids)	Arithmetic Intensity
(1)	Laplacian	8	2	1.00
(2)	Laplacian GSRB	29	2	3.63
(3)	Wave	16	3	1.33
(4)	Blur	31	2	3.88
(5a)	AWP-ODC: UXX	20	5	1.00
(5b)	AWP-ODC: ALLQ	181	33	1.37



(a) **Laplacian**
3-D 7-point stencil

(b) **Laplacian GSRB**
3-D 25-point stencil

(d) **"Wave" Stencil**
3-D 13-point stencil (2 time steps)

(c) **Blur Filter Stencil**
2-D 9-point stencil

Fig. 1. The structures of the example stencils

that individual stencil computations within one sweep do not depend on each other. This gives us the freedom to choose the grid traversal which works most efficiently for the stencil under consideration and the target machine. Cache blocking is a typical strategy which can be integrated most unobtrusively into existing code. It subdivides the domain into smaller subdomains (blocks), and the block sizes are chosen such that grid points which can be reused by stencil computations occurring later during the sweep are guaranteed not to be evicted from the cache before the actual reuse happens. More sophisticated methods extend the data reuse to the temporal dimension, e.g., by doing multiple sweeps across subdomains so that freshly computed values can be reused immediately for the computation of the next time step before they are written back to main memory [6], [7], [8], [9], [10], [11]. However, because the temporal loop carries dependences, such methods are more involved and also not as easy to integrate. Yet, such temporal blocking methods effectively increase the arithmetic intensity and therefore can push the stencil kernel into the compute bound regime.

III. RELATED WORK

As stencil computations map well to current multicore and many-core architectures, there has been a considerable research interest in the topic recently. In particular, there are a number of tools and frameworks targeting GPUs and multi-GPU systems (Mint in [12], Physis in [13], and methods optimized for GPUs [14], [15]).

Equally, work has been done in the CPU domain; recently there has been a special interest bandwidth-saving algorithm, in particular in temporal blocking and cache-oblivious methods [8], [16], [17], [18], [19], [10], [20]. In particular, Pochoir [9] is a special purpose compiler, which translates a C-based iterative stencil specification into a cache-oblivious implementation.

In a broader context, such code optimization techniques are investigated in compiler research, e.g., tiling of perfectly and imperfectly nested loops, e.g., in the polyhedral model [21], [22], [23]. On the other hand, advances in performance-oriented DSLs have been made in the projects around the Delite [24] DSL infrastructure.

In contrast to the cited code optimization research work, PATUS tries to provide a basis for (static) stencil-specific bandwidth-saving algorithms by means of a programming model-agnostic abstraction layer, which allows implementation of such algorithms as PATUS Strategies. Also, since hardware evolution is fastpaced, we aim at making the tool flexible enough that it can support a spectrum of platforms.

Previously we have shown that PATUS can accelerate single stencil kernels [2]; in this paper we use the tool to accelerate full applications.

IV. THE PATUS INFRASTRUCTURE

Currently, in most production environments, the burden of transforming a portable code (e.g., written in ANSI C or Fortran, which are the prevalent languages in scientific computing) into a code adapted to the target hardware platform is laid on the compiler. Although there have been huge advances in compiler technology, algorithm transformations, e.g., transforming an implementation into a bandwidth-saving one by introducing optimal cache blocking or even a form of temporal blocking, or, in fact, efficient automatic parallelization, are still beyond the reach of today's state-of-the-art compilers.

If we focus on a specific problem domain—such as stencil computations—we believe that special-purpose compilers are a viable solution: the problem can be expressed in a concise DSL, and the compiler can apply more aggressive, domain-specific optimizations, which might not be generally applicable. Thus, such an approach is able to bridge programmer productivity and code performance, for which generality is sacrificed.

In PATUS, the DSL approach is leveraged as a method of facilitating high-level abstraction specifications of stencil computations and for aiming at productivity and performance on current and future multicore and many-core platforms. With PATUS, we also try to address the portability issue, both of code and performance. The modular architecture of the backend system and the ability to specify common hardware characteristics in a configuration file make it possible to add backends for additional programming models and future hardware platforms. Thus, the same stencil specification can

be reused to generate code for any of the supported target architectures. Currently, x86 CPU architectures are supported (there are definitions for both SSE and AVX instruction sets) and, as a proof of concept, CUDA-programmable GPUs. In the case of CPUs we concentrate on shared-memory systems and use OpenMP as a programming model for parallelization. The CUDA backend works, but still more work is required to be able to compete against hand-coded CUDA code; hence, in this paper, we focus on CPU systems.

The performance portability aspect is addressed by employing the auto-tuning methodology. Parameters and variants of generated stencil implementations are tuned to deliver the optimal or a near-optimal performance for the particular stencil instance on the particular target hardware.

The conceptual tools to reach the goals of portability and reusability are separation and composability: PATUS separates the pointwise stencil evaluation from the implementation of stencil sweeps, i.e., the way the grid is traversed and the way the computation is parallelized. The idea is to support different types of bandwidth- and synchronization-saving algorithms for any concrete stencil instance. To this end, PATUS defines *Strategies* as an additional abstraction layer for grid traversal algorithms, into which the actual stencil computation is inserted. Strategies serve as algorithm templates, and by aiming at a high level of abstraction by leveraging DSLs once again, we make them not only independent of the concrete stencil instance, but also independent of the target programming model and target hardware. A description of the hardware architecture (in the form of an XML configuration file) then actually maps the algorithm to the specific target. PATUS comes with a selection of Strategies, which have proven successful in practice.

The idea of leveraging the auto-tuning methodology is to find the Strategy that yields the best performance (GFlop/s, GFlop/s per Watt, Joules, or any other performance metric which can be measured in the benchmarking code), for a fixed, application-specific stencil and a fixed hardware platform—the one the code will run on. Strategies are designed to be independent of the hardware architecture (which means that the algorithm runs on any platform), but they obviously have an impact on the performance as they define how data and computation is mapped to the compute units, and they can contain architecture-related optimizations.

The actual pointwise stencil computation (dictated by the application and therefore to be implemented by the user) is described by the *stencil specification*. As an example, Listing 1 shows the specification for the "Wave" stencil.

```
stencil wave {
  domainsize = (2..x_max-3, 2..y_max-3, 2..z_max-3);
  t_max = 1;
  operation (float grid u, float param dt_dx_sq) {
    float c1 = 2 - 15/2*dt_dx_sq;
    float c2 = 4/3*dt_dx_sq; float c3 = -1/12*dt_dx_sq;
    u[x,y,z; t+1] = c1*u[x,y,z; t] - u[x,y,z; t-1] +
      c2*(u[x+1,y,z; t] + u[x-1,y,z; t] + ...)+ //etc.for y,z
      c3*(u[x+2,y,z; t] + u[x-2,y,z; t] + ...);
  }
}
```

Listing 1. The stencil specification for the "Wave" stencil.

The actual stencil computation is defined within the `operation`. The arguments to the `operation` are the grids needed for the computation and application-specific parameters, should the calling code want to supply any precalculated values. Here, we need only one grid, and the stencil expects the discretization-dependent parameter corresponding to $\left(\frac{\delta t}{\delta x}\right)^2$.

The body of the `operation` contains the localized, pointwise stencil expression; the spatial stencil sweeps are not programmed explicitly, as explained above. The stencil specification language allows implementing more than one stencil expression in the `operation`. It also supports assignments to temporary variables, which can be reused in later stencil expressions. In this case, this feature is used to compute coefficients `c1`, `c2`, `c3` which PATUS recognizes as constants and hence hoists them out of the spatial loop.

Listing 2 shows an example of a PATUS Strategy: a cache blocking strategy. It is the one that will be used for the performance benchmarks shown in section V.

```
strategy cacheblocking(domain u,auto dim cb,auto int chunk){
  for t = 1 .. stencil.t_max { // iterate over time steps
    // iterate over subdomain
    for subdomain v(cb) in u(:; t) parallel schedule chunk {
      // calculate the stencil on each subdom. point
      for point p in v(:; t)
        v[p; t+1] = stencil (v[p; t]);
    }
  }
}
```

Listing 2. A PATUS cache blocking Strategy.

Within the `t` loop, the Strategy iterates over all the time steps. `stencil.t_max` refers to the respective declaration in the stencil specification and is a placeholder that can potentially also be replaced by a stopping criterion, although it is not supported as of yet.

Within one time step, the Strategy iterates in blocks `v` of size `cb` over the "root domain" `u`, i.e., the entire domain to which the stencil is applied. In this way, cache blocking is implemented. The `auto` specifier to the argument `cb` means that this parameter will be subjected to the auto-tuner. Both the root domain and `cb` are given as Strategy parameters.

Strategies also describe how the computation is parallelized (currently only within the shared-memory domain). In Listing 2, a blockwise parallelization is performed: by virtue of the `parallel` and the `schedule` keywords, `chunk` block-shaped subdomains `v` at a time are dealt out to each worker thread in a cyclic fashion. After the parallel `for` loop there is an implicit barrier. Finally, in the inner `p` loop, the formal stencil call applies the stencil to each point within `v`.

A. The Software Architecture

PATUS is built from four core components as shown in the high-level overview in Figure 2: the parsers for the two input files, the stencil definition and the Strategy, the actual code generator, and the auto-tuner.

The internal representation of the stencil specification, which is created by the stencil specification parser, consists of the domain size and number-of-iterations attributes and a

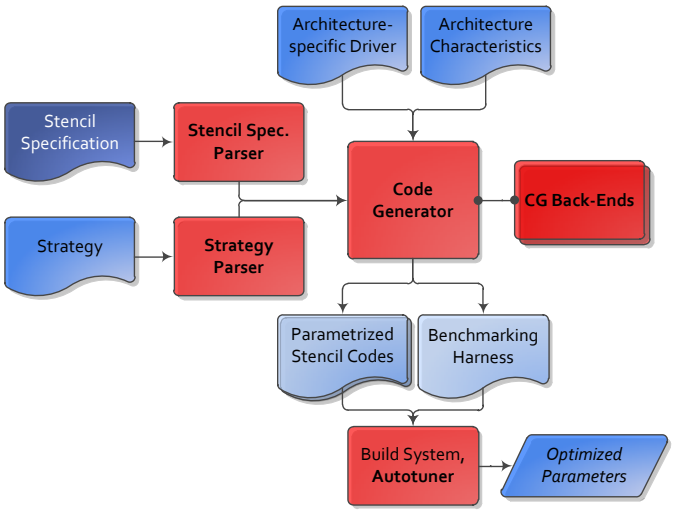


Fig. 2. A high-level overview of architecture of and the work flow of PATUS.

graph representation of the actual stencil parts described in the stencil `operation`.

The Strategy parser transforms the Strategy code into an internal representation based on the Cetus [25] compiler infrastructure. Both the stencil specification and the Strategy parsers were implemented using Coco/R [26].

The objective of the code generator is to translate the Strategy, into which the stencil was substituted, into the final C code. In particular, it transform Strategy loops into C loops and parallelizes them according to the specification in the Strategy. If desired, innermost loops containing the stencil computation are unrolled and vectorized. Code generation backend specializations for specific architectures and programming models take care of generating the correct idioms for index calculations, vectorization, and parallelization. The "architecture characteristics" input is an XML configuration file, which describes aspects of the target platform including the number of levels of parallelism, how to map arithmetic operators to SIMD intrinsics for vectorization, and which code generation backend to use.

Along with an implementation for the stencil kernel, the code generator also creates a benchmarking harness from the "architecture-specific driver." The latter is a programming model-specific template into which the dynamic memory allocations, the grid initializations, and the kernel invocation are substituted.

B. Code Optimization Techniques

The overall code generation process is driven by the selected Strategy, which defines the structure of the generated code, including the parallelization mechanism. However, there are a number of optimizations which are not controlled by the Strategy:

1) Loop unrolling: This well-known compiler optimization replicates the loop body to decrease the loop overhead and potentially increase instruction parallelism, e.g., by breaking dependence chains. However, unrolling loops increases register pressure and code size, which might lead to L1 instruction cache misses. Hence the optimization has to be applied carefully. By default, PATUS generates several code variants with different unrolling configurations of the innermost loop nest containing the stencil computation. The code variants are added to the auto-tuner search space.

2) Vectorization: Vectorization has proven to be the key optimization for numerous stencils, especially in view of the growing SIMD vector width.Using the hardware's SIMD units in an optimal way is critical for performance; even if we trust the compiler to vectorize scalar code we could observe that explicitly vectorized code using SIMD intrinsics yields significantly better performance. This is mostly due to the fact that a general purpose compiler has to err on the safe side and applies vectorization conservatively.

PATUS can generate code using explicit SIMD intrinsics, and by default it is assumed that the unit stride direction is padded to a multiple of the SIMD vector length. The benefit of this is that in the non-unit stride dimensions no loads are needed which are not aligned at SIMD vector boundaries, as unaligned loads may incur a latency penalty or may not even be supported on other architectures and therefore require a workaround. The intrinsics are not hard-coded into the code generator, but rather are defined in the architecture description. In the configuration, mappings from standard arithmetic operators to intrinsic functions, as well as from arbitrary functions to their SIMD equivalents, can be defined. Thus it was straightforward to add support for AVX,— or, in fact, for any other SIMD instruction set.

3) Inline assembly code generation: Recently, a code generation module was added to PATUS that translates the innermost loop and the therein contained stencil computation into inline assembly rather than into C code. The main benefits are more compact SIMD code and more efficient index calculations.

If possible, the addresses of all the grids which are read or written during a stencil computation are kept in registers, as well as all the index offsets required to access values neighboring grid points. If there are sufficiently many registers available which can hold all of the necessary offsets from the center point, the index computation can be expressed by a single address operand.

Loop unrolling has a special importance in this context. When loop unrolling is turned on for the dimension corresponding to the innermost loop, each compute instruction is replicated so that there are as many parallel strands of computation as the number by which the loop is unrolled. This helps to increase instruction level parallelism and typically results in increased performance, provided that there are sufficiently many SIMD registers to hold the operands. Optimal instruction scheduling to further potentially increase the instruction level parallelism is work in progress.

Register allocation is done by running a graph coloring algorithm on the live analysis graph of the instruction list. So far, PATUS implements only a greedy coloring, which orders the vertices by descending degree and assigns each vertex

in the list the first available color, or a new one if none is available. The allocation quality of this approach seems to be sufficiently good for our case.

Again, as for the intrinsics, mnemonics are not hardcoded into the code generator, but are defined in the architecture description. The code generator creates a generic list of instructions which use nondestructive operands (which is supported by AVX). This generic list is then translated to the actual mnemonics, and the necessary conversions are made if the target is defined to support only destructive forms (SSE, for instance).

C. Auto-Tuning

In the context of PATUS, the auto-tuning search space is the Cartesian product of the value range of Strategy parameters and code variants generated by the code generator, e.g., various loop unrolling configurations. The PATUS auto-tuner is driven by some direct search method (e.g., greedy, simplex search, a genetic algorithm) to navigate the search space, as an exhaustive search is not feasible. The performance results shown in section V were all obtained using the genetic algorithm.

Ideally, the search space would be reduced by discarding parameterizations which are known to yield suboptimal performance results, e.g., by predicting the performance using an analytical model. In the case of PATUS, such a model would have to be Strategy-specific—the reason why we have only taken the simple approach of trying to extract optimal or near-optimal data from the entire search space so far.

D. Usage, Integration, and Application-specific Tuning

PATUS is written in Java and the only other external dependency is the open-source computer algebra system "Maxima." PATUS generates C functions, which can be called directly from a C or Fortran application code.

The generated benchmarking harness includes a Makefile with a specific auto-tuning target. Invoking this target with application-specific parameters (e.g., the domain size) starts the PATUS auto-tuner, which writes the best parameter values to a header file. Thus, whenever the generated code is called, it uses these parameter values found by the auto-tuner, and only application-specific parameters have to be passed to the function.

To integrate PATUS-generated code into AWP-ODC, we used the mode in which PATUS processes a foreign-language input (Fortran, in this case) with PATUS-specific sections (the stencil specifications), in order not to clutter the source code structure. In this mode, PATUS extracts the designated fragments, interprets them as input, and generates the corresponding optimized stencil implementation. The PATUS specifications in the original file are replaced by calls to the generated functions. This automatically preprocessed code can then be compiled with a regular compiler.

In practice, this meant translating the target stencils into PATUS specifications in the original source files, adding the preprocessing step to the build toolchain (i.e., the Makefile), and invoking the auto-tuner per generated stencil, providing the application's local problem size as arguments.

V. PERFORMANCE BENCHMARKS

A. Experimental Testbeds

1) AMD Opteron Interlagos: AMD's Opteron Interlagos CPUs are based on the recent, newly designed "Bulldozer" microarchitecture. The chips are manufactured in 32-nm silicon on insulator high-κ metal gate process technology. Each Interlagos socket contains two dies, each of which in turn contains four so-called "modules." Each module contains two separate full out-of-order integer cores. However, it contains only one floating point unit. The latter features two 128-bit SSE/AVX engines, which can be combined into one 256-bit engine to support the AVX-256 instruction set, including fused multiply-add (FMA4) instructions.

Each integer core has its own 16-KB four-way L1 data cache. The 64-KB two-way L1 instruction cache, however, is again shared among the cores within a module. 8 MB of L3 cache are shared among the four modules on a die.

Each die has its own memory controller. Thus, one Interlagos socket is inherently a NUMA architecture. We used the two sockets available on one node, which is part of a Cray XE6 system. Thus, we have 32 hardware threads at our disposal. The available bandwidth to DRAM was measured (with STREAM Triad [27]) to be around 44 GB/s when both memory controllers on both sockets were used.

2) Intel Xeon Sandy Bridge: Intel's recent Sandy Bridge microarchitecture (manufactured in a 32 nm process), which is based on the Intel Core and Intel Nehalem architectures, improves on the previous designs by supporting the 256-bit AVX instruction set, using a decoded instruction cache, and doubling the internal bandwidth (by using two "ports" for memory operations). Each of the two Xeon E5-2670 Sandy Bridge sockets contains eight superscalar out-of-order cores. Thus, 16 threads were available, as Hyper-threading was not activated on the system.

In contrast to the AMD Interlagos, the Intel architecture has distinct units for 256-bit SIMD floating point add and multiply, respectively. Thus, the microarchitecture is superscalar only for add-multiply mixes.

Each core is equipped with 32 KB of L1 instruction cache and 32 KB L1 data cache, as well as a 256-KB unified L2 cache. All the cores share 20 MB of L3 cache. Using the STREAM Triad benchmark, we observed about 62 GB/s of bandwidth to DRAM on one dual-socket compute node.

B. Kernel Benchmarks

In this section, we show performance results obtained by PATUS-generated codes on both recent AMD and Intel microarchitectures. We also contrast the performance achieved by PATUS with the performance of corresponding reference codes compiled with a range of state-of-the-art compilers. The PATUS codes were all compiled with the GNU C compilers as the other compilers do not support either SIMD intrinsics or inline assembly code. The benchmarks were all done in single precision for 16M-sized domains, i.e., 256^3 and 4096^2 grid points in 3-D and 2-D, respectively. The choice of the precision

mode was given by the AWP-ODC application, which uses this mode. Thus, to make the data comparable we used single precision for all of the benchmarks. The double precision variants perform at half of the numbers shown.

1) Performance Comparison on AMD Interlagos: Figure 3(a) shows single precision performance and scalability results for the example stencils on a dual-socket AMD Opteron Interlagos node, i.e., for 1 up to 32 hardware threads. For testing the scalability, we filled modules first (i.e., the results for 2 threads were obtained by using the two cores in one module), then sockets. In general, when looking at the entire bars, almost linear scalability can be observed, except when going from 1 to 2 threads, which shared the floating point unit of one module.

By means of the differently colored parts of the bars, the figure shows the performance reached with particular vectorization techniques. While the vectorization and SIMD code generation methods were varied, all results used the cache blocking Strategy shown in section IV with tuned parameters as well as the best loop unrolling configuration. The red parts show the results when the code was not explicitly vectorized, i.e., any vectorization was done solely by the compiler (gcc 4.6.2). The orange parts show the performance increase gained by employing explicit vectorization using SSE intrinsics, which were generated by PATUS. In most of the cases it can be seen that this about doubles the performance, except in the case of the 7-point Laplacian, which is severely bandwidth-limited, and thus the maximum performance, which is indicated by the black markers, is already largely reached in the scalar code. The 3-D stencils are largely able to make full use of the memory bandwidth; for the 2-D case (the Blur stencil), however, evidently there is still more optimization potential. Also, PATUS is not yet able to fully optimize very large stencils such as ALLQ (which in fact consists of 19 stencil expressions). This particular kernel also contains many floating point divisions, which incur high latencies.

In Figure 3(b), we compare the performance reached by compiling a basic nonoptimized reference C code with various compilers against the PATUS-generated code compiled with gcc 4.6.2. For all the compilers we used the optimization flags recommended by the computing center and the compiler vendors (at least `-O3`). The figure shows that most of the PATUS-generated codes can achieve a higher performance than current C compilers especially for higher thread counts. On this platform, the Cray C compiler outperforms the other compilers in almost all of the cases—mostly due to its excellent vectorizing capability—and beats PATUS in the case of the 2-D Blur stencil. In the reference codes, no NUMA optimizations were done. This becomes obvious (and detrimental to performance) when scaling past one NUMA domain: The performance of the reference codes breaks down while the NUMA-aware PATUS codes continue to scale.

2) Performance Results on Intel Sandy Bridge: Figure 4(a) shows the single precision performance numbers for the individual vectorization methods on the Intel platform. As the Sandy Bridge has distinct 256-bit adders and multipliers, we did not include any AVX-128 results here. Again, cores were filled before sockets; we used LIKWID [28] to do this. As on the AMD architecture, in all of the cases, explicit vectorization improves the performance significantly over the compiler-vectorized code, except when the memory bandwidth is already exhausted by the basic version (in the Laplace stencil with 8 and 16 threads). The black markers again show the maximum reachable performance as dictated by a kernel's arithmetic intensity and the available bandwidth. As can be seen, the Laplace stencil indeed reaches this limit as soon as 4 or more cores are used. Due to the Xeon's greater bandwidth, the larger stencils become compute bound in theory for low thread counts (the higher-order Laplacian and the Blur stencil for 1 and 2 threads, the Wave stencil and ALLQ for 1 thread). The upper per-core performance bound was computed taking into account the superscalar architecture; in an addition-multiplication mix, every 0.38 cycles an instruction can be issued, on average.

In Figure 4(b), the contribution of individual optimization techniques is shown for the vectorization method which yields the best performance. The red parts show the performance that can be reached without explicit vectorization and without applying cache blocking and loop unrolling. Turning on vectorization gives a substantial performance increase in most of the cases, which means that gcc did not manage to vectorize the code efficiently. The yellow and green parts, which correspond to the contribution of cache blocking and loop unrolling, respectively, show that, depending on the stencil, both optimizations are viable. Larger 3-D stencils benefit from cache blocking, while for the 2-D stencil, which does not do memory accesses which are far away from the current position in the linearized array, loop unrolling is essential. The overlaid black markers show reference code performance compiled with the compilers available on the system. Here, we included a version that forced the Intel compiler to vectorize (by manually inserting a "#pragma simd" into the code).

C. Performance and Scalability of a Wave Equation Solver

Figure 5 shows both strong and weak scaling in log-log plots of a wave equation solver, which essentially extends the wave stencil (example 3 in section II) to the distributed-memory domain using the hybrid MPI + OpenMP programming model, for up to 4096 MPI processes on a Cray XE6, which features AMD Opteron Interlagos processors.

Each MPI process spans an entire NUMA domain, i.e., uses 8 OpenMP threads; thus, in effect, 16 (one socket) up to 32,768 cores were used. We deliberately placed one MPI process per NUMA domain for a fair comparison against the original code, since in this case no NUMA optimizations are required for scalability. We compare PATUS against the Cray C compiler, the best compiler on this platform (cf. section V-B1).

The strong scaling test [Figure 5(a)] was done with a domain of size 1024^3. For the largest per-process domains, PATUS achieves a speedup of up to $1.9\times$ over the Cray C compiler. As the local domain sizes decrease, the speedup drops, mostly

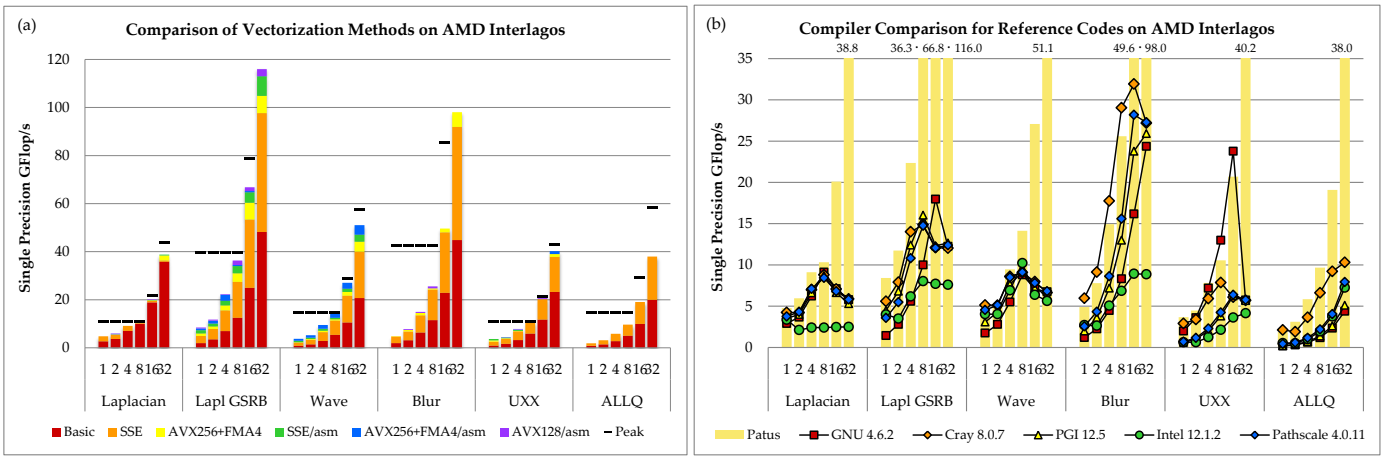


Fig. 3. **(a)** Single precision performance results and scalability of different vectorization modes for the example stencils on AMD Opteron Interlagos. (Basic: vectorization done by the compiler (gcc 4.6.2), SSE: explicit PATUS-generated SSE intrinsics, AVX256+FMA4: explicit AVX and FMA4 intrinsics using 256-bit wide SIMD vectors, AVX256+FMA4/asm: inline assembly mode using AVX-256 and FMA4 instructions, AVX128/asm: inline assembly mode using the AVX instruction set with 128-bit wide SIMD vectors.) **(b)** Performance comparison of basic reference stencil codes compiled with various compilers vs. PATUS-generated code.

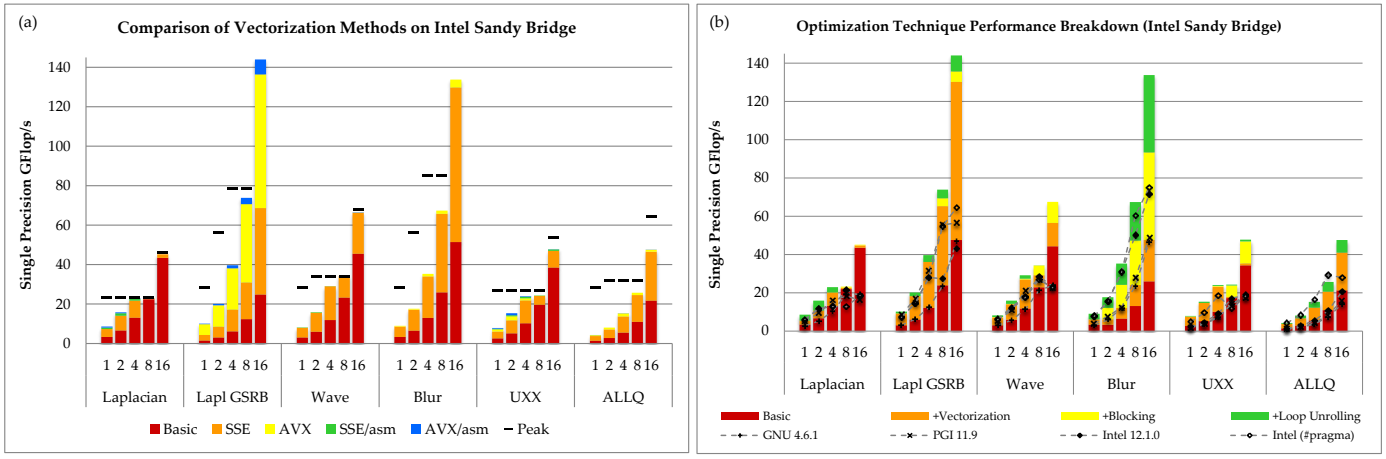


Fig. 4. **(a)** Single precision performance results and scalability of different vectorization modes for the example stencils on Intel Xeon Sandy Bridge. (Basic: vectorization done by the compiler (gcc 4.6.1), SSE: explicit generated SSE intrinsics, AVX: explicit AVX-256 intrinsics, SSE/asm: inline assembly mode using SSE instructions, AVX/asm: inline assembly mode using the AVX-256 instruction set.) **(b)** The color bars show the contribution to the performance of individual optimizations (vectorization, cache blocking, loop unrolling) for the example stencils on the Intel Xeon Sandy Bridge architecture. The black markers show the performance of the reference stencil codes compiled with various compilers. In the "Intel (#pragma)" version, an explicit pragma was manually inserted to force vectorization.

due to the cache optimizations becoming less effective: also the original code can utilize the cache efficiently for small local problem sizes. The speedup still remains above 1. This is visualized by the blue line. The figure shows that scaling this code beyond 512 processes is not ideal for this problem size (ideal scaling is shown by the dashed line). As the local domain sizes drop below 128^3, communication becomes the bounding factor, although the code overlaps computation and communication. This can be observed more clearly in the weak scaling graphs in Figure 5(b): Small local 64^3-sized domains lead to sub-optimal scaling, whereas with local domains of size 512^3 perfect scaling can be achieved. Also, the speedup obtained with PATUS over the Cray C compiler stabilizes in this case at around $1.7\times$.

In terms of absolute performance, in the 512^3 case, the original code ran at 22.2 TFlop/s on 1024 nodes of the Cray XE6. Using PATUS, the performance is improved to 38.2 TFlop/s.

D. Large-Scale Earthquake Simulations: AWP-ODC

For this benchmark, we use the anelastic wave propagation code AWP-ODC with integrated PATUS-generated kernels. The measurements were carried out on a cluster with Intel Xeon Sandy Bridge E5-2670 CPUs. We used the pure-MPI version of AWP-ODC, which was shown to be scalable up to hundreds of thousands of nodes [1]. We did not touch any of the original MPI communication code, but used PATUS to tune parts of the actual local computation. The code was compiled with the Intel Fortran compiler 12.1.0, and

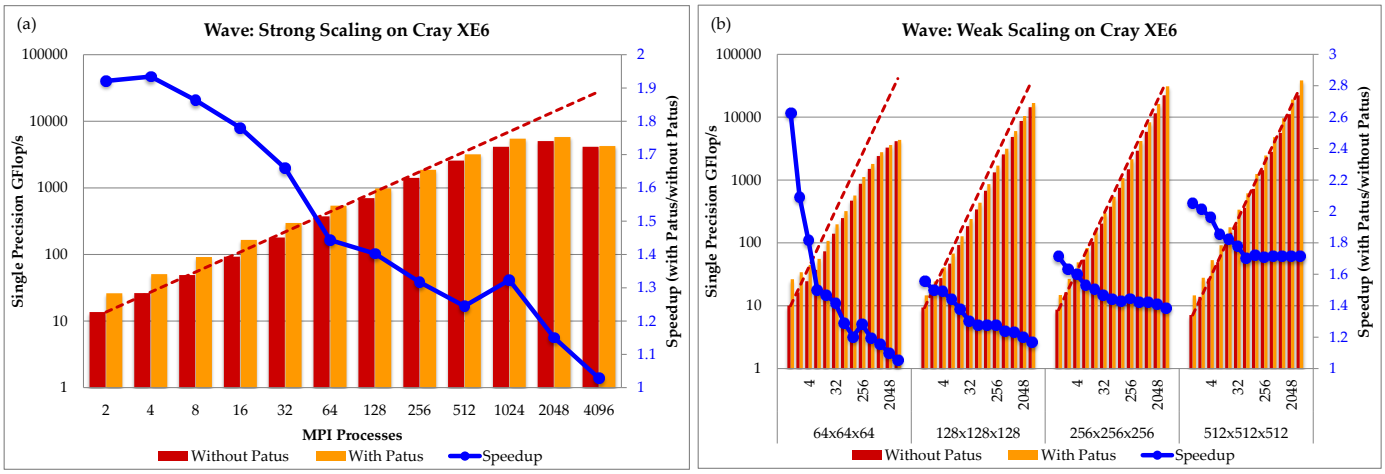


Fig. 5. **(a)** Strong and **(b)** weak scaling of the Wave equation solver on a Cray XE6 (AMD Opteron Interlagos processors). Note that the figures are log-log plots. Red bars show the performance of the original code, orange bars show the performance when the PATUS-generated code was built in. The dashed lines show perfect scaling behavior. The blue lines on the secondary y-axis are the speedup of PATUS vs. the original code.

OpenMPI 1.4.1 running on the InfiniBand fabric was used as MPI implementation. The kernels generated by PATUS were compiled with gcc 4.6.1.

We ran both the elastic and the viscous modes of the AWP-ODC code. In both modes, the UXX-type kernels update the velocity field, in which around 30% of the compute time is spent, whereas the stress tensor field is updated by a set of smaller kernels in the elastic mode (comprising around 45% of the computation time) and the one large kernel ALLQ in viscous mode. For stencil computations, the Fortran compilers are able to optimize much better than the C compilers, and the Fortran version of ALLQ actually performed better than the PATUS-generated version. Hence, we used the original Fortran version for ALLQ.

The results shown in Figure 6 are weak scaling results for 1 up to 512 MPI processes using local problem sizes of 256^3 grid points. The y-axis shows the time in seconds required to complete one time step. We used 8 of the 16 available cores per node, distributing the processes equally among the two sockets (again, LIKWID [28] was used to ensure that MPI processes were pinned to CPU cores), since tests had shown that using all 16 cores was detrimental to performance due to the L3 cache being shared among all cores. In fact, the figure shows that even when increasing the number of MPI processes from 1 to 2 and 4, the shared resources have a negative impact on the performance. However, when running across multiple nodes (i.e., with more than 8 processes), the code exhibits excellent scalability (apart from the spike occurring in viscous mode with 256 MPI processes which, for some reason, occurred throughout all of the 3 runs we performed).

When running one or two processes per NUMA domain, the PATUS version achieved a speedup of 15% and 7% on average in the elastic and viscous modes, respectively. In the other case, up to 6% and 4% speedup were achieved in the elastic and viscous modes, respectively (disregarding the spike).

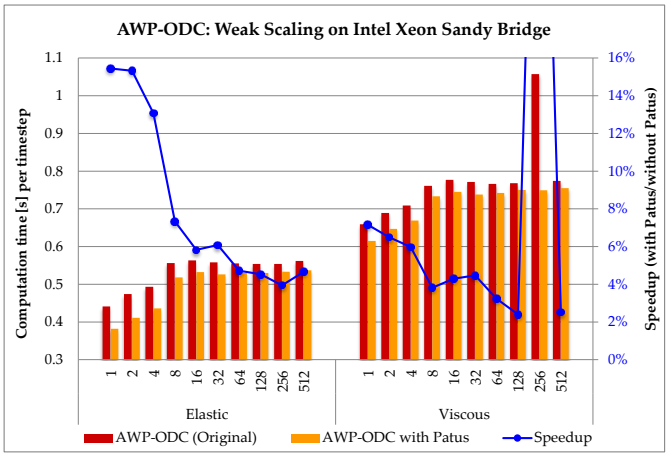


Fig. 6. Weak scaling of the AWP-ODC code from 1 to 512 MPI processes

VI. CONCLUSION

In this paper, we presented an approach to automatically generate and optimize stencil kernels, including stencils originating from a real-world application: the anelastic wave propagation code AWP-ODC. We showed that PATUS is able to generate scalable threaded implementations for the individual stencil kernels leveraging nontrivial parallelization and code optimization strategies—including cache blocking, explicit vectorization, and loop unrolling,—as well as the auto-tuning methodology for performance portability.

In the kernel benchmarks, we have shown that vectorization is an essential component for performance optimization. With vectorization implemented efficiently, we were able to double the performance over the compiler-generated vectorization (taking the GNU C compiler as a base). Commercial compilers are able to vectorize more efficiently, but even using state-of-the-art compiler technology, naïvely programmed stencil codes cannot match the performance of the generated and auto-tuned

codes, especially for higher thread counts.

Using PATUS, the implementation of a stencil kernel is less involved than a naïve C implementation (e.g., the programmer does not have to worry about array index calculations, which become necessary as C does not support multidimensional dynamically allocated arrays), yet PATUS makes sure that the performance-critical optimizations are performed automatically, thus increasing programmer productivity.

Our wave equation solver achieved 38 TFlop/s on 1024 nodes of a Cray XE6, which is a 70% performance improvement resulting from applying PATUS, with the potential to save millions of core hours for this type of heavily used community application.

Integrating PATUS into the full AWP-ODC application, we were able to achieve up to 15% performance improvement provided that only a limited number of MPI processes per NUMA domain were used. Using more cores put more stress on the last level cache, which results in a performance penalty. This result also shows that it is important to embrace a threaded model, especially as we anticipate that the core count will rise tremendously as we approach the exascale.

In both cases, the wave equation solver and AWP-ODC, we could show that we could employ PATUS successfully and observe a speedup over the original code, which is valuable since any speedup in such applications, which typically perform tens of thousands of time steps and consume millions of system units, effectively reduces the time to solution and cost.

PATUS can be obtained under the LGPL license and is accessible at http://code.google.com/p/patus.

ACKNOWLEDGMENT

We acknowledge the Swiss National Supercomputing Center and Intel's Parallel and Distributed Solutions Division for giving us access to their hardware and for their support. The work done in Switzerland is funded through the *Petaquake* project under the HP2C (High Performance and High Productivity Computing) initiative.

REFERENCES

[1] Y. Cui, K. Olsen, T. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. Panda, A. Chourasia, J. Levesque, S. Day, and P. Maechling, "Scalable Earthquake Simulation on Petascale Supercomputers," in *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, 2010, pp. 1–20.

[2] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures," in *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2011)*, 2011, pp. 1–12.

[3] M. Christen, "Generating and Auto-Tuning Parallel Stencil Codes," Ph.D. dissertation, University of Basel, Switzerland, 2011.

[4] K. Olsen, "Simulation of Three-Dimensional Wave Propagation in the Salt Lake Basin," Ph.D. dissertation, Univ. of Utah, 1994.

[5] C. Marcinkovich and K. Olsen, "On the Implementation of Perfectly Matched Layers in a 3D Fourth-Order Velocity-Stress Finite-Difference Scheme," *J. Geophys. Res.*, vol. 108 (B5), 2003.

[6] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart, "Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures," in *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2009)*, May 2009, pp. 1–10.

[7] M. Frigo and V. Strumpen, "The Cache Complexity of Multithreaded Cache Oblivious Algorithms," in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2006)*, 2006, pp. 271–280.

[8] R. Strzodka, M. Shaheen, D. Pajak, and H. Seidel, "Cache oblivious parallelograms in iterative stencil computations," in *Proc. ACM Int'l Conference on Supercomputing (ICS 2010)*, 2010, pp. 49–59.

[9] Y. Tang, R. Chowdhury, B. Kuszmaul, C. Luk, and C. Leiserson, "The Pochoir Stencil Compiler," in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011)*, 2011, pp. 117–128.

[10] J. Treibig, G. Wellein, and G. Hager, "Efficient Multicore-Aware Parallelization Strategies for Iterative Stencil Computations," *J. Comp. Sci.*, vol. 2, no. 2, pp. 130 – 137, 2011.

[11] D. Wonnacott, "Time skewing for parallel computers," in *Proc. Int'l Workshop on Compilers for Parallel Computing (CPC 1999)*. Springer-Verlag, 1999, pp. 477–480.

[12] D. Unat, X. Cai, and S. Baden, "Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C," in *Proc. ACM Int'l Conference on Supercomputing (ICS 2011)*, 2011, pp. 214–224.

[13] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers," in *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011)*. IEEE Computer Society, 2011.

[14] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations," *Int. J. Parallel Prog.*, vol. 39, pp. 115–142, February 2011.

[15] Y. Yang, H. Cui, X. Feng, and J. Xue, "A Hybrid Circular Queue Method for Iterative Stencil Computations on GPUs," *J. Comp. Sci. Tech.*, vol. 27, pp. 57–74, 2012.

[16] Z. Li and Y. Song, "Automatic Tiling of Iterative Stencil Loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, pp. 975–1028, 2004.

[17] K. Datta, "Auto-tuning Stencil Codes for Cache-Based Multicore Platforms," Ph.D. dissertation, Univ. of California, Berkeley, 2009.

[18] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An Auto-tuning Framework For Parallel Multicore Stencil Computations," in *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[19] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs," in *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*. IEEE Computer Society, 2010.

[20] T. Malas, A. Ahmadia, J. Brown, J. Gunnels, and D. Keyes, "Optimizing the Performance of Streaming Numerical Kernels on the IBM Blue Gene/P PowerPC 450 Processor," *IJHPCA*, 2012.

[21] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. Strout, "Parameterized Tiled Loops for Free," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, 2007, pp. 405–414.

[22] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, 2008.

[23] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. Khan, "Loop Transformation Recipes for Code Generation and Auto-Tuning," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, G. Gao, L. Pollock, J. Cavazos, and X. Li, Eds., vol. 5898. Springer Berlin / Heidelberg, 2010, pp. 50–64.

[24] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A Heterogeneous Parallel Framework for Domain-Specific Languages," in *Proc. Int'l Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, 2011, pp. 89–100.

[25] H. Bae, L. Bachega, C. Dave, S. Lee, S. Lee, S. Min, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," in *Proc. Int'l Workshop on Compilers for Parallel Computing (CPC 2009)*, 2009.

[26] H. Mössenböck, M. Löberbauer, and A. Wöß, "The Compiler Generator Coco/R," http://www.ssw.uni-linz.ac.at/coco, accessed April 2012.

[27] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," Department of Computer Science, Rutgers University, Tech. Rep. DCS-TR-379, 1998.

[28] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance-oriented Tool Suite for x86 Multicore Environments," in *Proc. First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI2010)*, San Diego CA, 2010.